Software Tech News

The DoD Source for Software Technology Information.

Vol. 3- No. 4
Software Reliability

In This Issue:

www.dacs.dtic.mil/awareness/newsletters/listing.shtml

A Software Engineering Course for Trustworthy Software

by Larry Berstein, Have Laptop - Will Travel

Trustworthy software always provides the same results to the same input. Trusted software must be achieved as people come to depend on software-based systems for their livelihoods and, as with emergency systems, their very lives. Software is fundamental to computerized systems, yet it is rarely taught as an entity whose trustworthiness can be improved with specific techniques.

A software engineering course is proposed to treat the issues of software trustworthiness. Feedback on the need and contents for such a course is sought. The Committee for National Software Studies sponsors a web page (www.cnsoftware.org) discussing trustworthy software and provides a means for commenting on the subject.

Software has a weak theoretical foundation, yet there exists a body of knowledge that is sometimes used to improve software trustworthiness. The reason for a system failure due to software is often due to something that could have been avoided with a different type of design. Unfortunately the State-of-the-Practice lags the State-of- the-Art. Pro. Shiu-Chin of Syracuse University writes that we should

"...develop...curricular support for...design methods...as a means to support system design. The level of professional practice will improve when we have practical high-assurance design methods which work-and when we train our students to use them."

Most current software theory focuses on its static behavior by analyzing source listings. There is little theory on its dynamic behavior and its performance under load.

$$R(t) = e^{-t/\alpha}$$

Continued on page 8

DoD Data & Analysis Center for Software
http://www.dacs.dtic.mil

Software Reliability

How Reliable are Requirements for Reliable Software?

by Herbert and Myron Hecht, SoHar Inc.

Introduction

Missing, inaccurate or incomplete requirements lead to errors in software development and usually also prevent these errors from being detected during the testing phase. Functional testing is based on the requirements; a missing or inaccurate one will not be detected. Structural testing is based on the developed code; an unstated requirement is unlikely to be implemented and will not be detected. Operational failures due to omissions or inaccuracies cause major economic losses or even casualties, and corrective measures are far more costly than they would be if the defect had been caught earlier. A distinguishing feature of *reliable software* is that it contains fault tolerance provisions, such as alternative exits when the assertions fail, roll-back and re-try, recovery blocks, or multi-version programming. In most cases these provisions prevent or attenuate the effect of hardware and software failures that would have occurred in their absence, but there have also been incidents where the fault tolerance objectives have not been achieved and the reasons for the failure have usually included missing or ill-formulated requirements.

In the body of this paper we first describe *what* is missing in requirements, then why it is missing, and after that we explore corrective measures and test strategies for verification of reliable software.

What is Missing in Requirements for Reliable Software ?

Difficulties in formulating requirements for reliable software frequently arise from inability to identify (a) all sequences that invoke fault tolerance provisions and (b) future operational environments. We discuss these in turn.

An analysis of failures in a telephone switching system paper notes that (a) the largest cause category (44% of failures) comprised combination hardware/ software faults. In most cases it was the inability of the software to recover from hardware faults that it was intended to protect against, and (b) that the faults leading to the most severe consequences "were introduced during the specification period and are therefore difficult to solve."[1] Similarly, a GAO report on serious problems in ten computer-based systems traces these to failure to implement "a process for disciplined, consistent procedures for software requirements management, quality assurance, configuration management, and project tracking."[2] Requirements management is the key since all the other functions depend on it.

The reasons for the deficiencies in requirements include disbelief that more than one failure can occur during an operating interval, or neglecting the possibility that a single event (e. g., a short power interruption) can trigger several fault responses in the system is frequently overlooked. Even where requirements for fault tolerance provisions are explicit, the designers may misinterpret them unless a specific review or consultation process is provided. This is seen in an experiment sponsored by NASA to investigate the independence of fault responses in redundant software.[3] The specifications for the program were very carefully prepared and then independently validated to avoid introduction of common causes of failure. Each programming team submitted their program only after they had tested it and were satisfied that it was correct. Then all 20 versions were subjected to an intensive third party test program. The objective of the individual programs was to furnish an orthogonal acceleration vector from the output of a non-orthogonal array of six accelerometers after up to three arbitrary accelerometers had failed. Table 1 shows the results of the third party test runs in which an accelerometer failure was simulated.

Table 1. Tests of Redundancy Management Software

No. of Prior Anomalies	Observed Failures	Total Tests	Failure Fraction
0	1,268	134,135	0.01
1	12,921	101,151	0.13
2	83,022	143,509	0.58

The number of rare conditions (anomalies) responsible for failure is one more than the entry in the first column (because a new accelerometer anomaly was simulated during the test run, and it is assumed that the software failure occurred in response to the new anomaly). In slightly over 99% of all tests a single rare event (accelerometer anomaly) could be handled as indicated by the first row of the table. Two rare events produced an increase in the failure fraction by more than a factor of ten, and the majority of test cases involving three rare events resulted in failure. Although the statement of the problem clearly required that up to three anomalies had to be tolerated, the software developers had difficulties in providing for the required response to more than a single malfunction. Also, the developers' own test scenarios did not sufficiently explore multiple failure conditions.

The second difficult area for requirements is the response to changes in the system environment. Computers and operating systems are periodically updated and new models of sensors or actuators may be introduced. The application program may be reviewed and tested for proper operation in the new environment, but safeguards to prevent use of the software in the wrong configuration are frequently missing. Thus, if a problem develops with the most recent release of the operating system and it is decided to revert to the previous one, the need to go back to the old application software may be overlooked. Several crashes of important programs have been attributed to such lapses in configuration management. Providing a version check as part of the initialization should be a mandatory requirement but apparently it is not.

Why Requirements are Incomplete

The primary cause of incomplete requirements is the waterfall model that assumes that requirements can be completely formulated at the outset for systems of any scale. That, coupled with a procurement system that discourages continuous updating of user needs, casts in concrete requirements that were developed under severe time constraints and many months, possibly years, before the development started.

In a large organization, and particularly in branches of the government, at least three entities participate in the formulation of requirements: the user, the funding agency, and the office in charge of the development. The first step in the process is a statement of operational needs generated by the user. This is typically forwarded to the developer for obtaining a budgetary estimate, and then the need and the estimate are submitted for funding. In favorable circumstances the funding will be approved, but usually after considerable delay. Once approval has been obtained, the emphasis is on avoiding further delay. Previously generated requirements are dusted off and only cursorily reviewed to determine that they really represent current needs.

Finally, we want to reiterate the difficulty of conceptualizing and understanding the effect of multiple failures that was already mentioned in the preceding section. The resource-constrained environment of a typical software development provides a further obstacle to evaluating whether the requirements fully cover all required combinations of failures.

Corrective Measures

In the two preceding sections we have seen that requirements for highly reliable systems may be incomplete, particularly with regard to the reliability related features. Missing or incomplete requirements are not likely to be identified by either functional or structural testing and thus tend to persist into the OPEVAL and usage phases, sometimes constituting safety hazards and always imposing a very high cost for correction in the late lifecycle phases.

Since we have identified the waterfall model as a root cause of incomplete requirements it is appropriate to mention techniques that recognize that requirements evolve during development. Among these are the spiral development model[4] and rapid prototyping.[5] Narrower techniques are summarized in Table 2.

As a baseline (against which corrective measures will be evaluated) let us assume that the software development proceeds in a disciplined manner, and that applicable techniques from the requirements engineering discipline have been used.[6]

Continued on page 4

How Reliable are Requirements for Reliable Software?

Continued from page 3

Table 2. Techniques for Avoidance of Incomplete Requirements

TECHNIQUE	BENEFITS
Formal Methods[7]	Can detect some inconsistencies and instances of incomplete requirements.
Condition Tables[8]	Very effective detection of incomplete requirements.
Scenario-Based Testing[9] Thread-Based Testing[10] Task-Based Testing[11]	All of these introduce elements of OPEVAL into the earlier test phases. Effectiveness depends on skill of the implementers.
Random Testing[12]	Multiple Random Number generators for groupings of exception conditions can detect missing requirements for combination events.

The first two entries in the above table address primarily logical gaps or inconsistencies. The three test methods that are grouped together in the next row go beyond the traditional requirements format and recognize the need for more user interaction with the development. Random testing has been shown to provide high coverage in the cited reference, but it needs an oracle to identify the correct test outcome where that is not obvious.

While user involvement during development will help, the typical task-oriented user does not recognize deficiencies in exception handling or the need for automated configuration monitoring. *Requirements elicitation* improves the effectiveness of user interaction but must be directed to areas where deficiencies are likely to exist. This requires knowledge of past failures and better utilization of existing databases for identifying the role of incomplete requirements. Thus collection and analysis of failure data emerges as the key to long term improvements for formulation of reliable requirements for reliable systems.

About the Authors

Herbert Hecht founded SoHaR in 1978 and is currently Chairman of the Board. Previously he held engineering management positions at The Aerospace Corporation and at Honeywell Flight Systems. His chief professional interest is the reliability and availability of computer based systems. He has served as a Governor of the IEEE Computer Society and as a visitor in Computer Engineering for ABET. Recently he has been on the National Research Council Committee that evaluated long term use of the International Space Station.

He earned BEE and MEE degrees from City College and Polytechnic University of New York, respectively, and received a Ph. D. in Engineering from UCLA.

SoHaR is an acronym for Software and Hardware Reliability.

Myron Hecht is co-founder and President of SoHaR Incorporated. His activities in basic research and development at SoHaR have resulted in new architectures for real time distributed systems, methodologies for the development and verification of fault tolerant software, and design techniques for highly reliable distributed systems for process control and C3I. In prior employment he developed and verified computer codes for nuclear power plants at SAIC and Westinghouse.

Mr. Hecht has an M.B.A, an M.S. in Nuclear Engineering, and a B.S. in Chemistry, all from UCLA. He is a member of the IEEE and has served its standards committees. He has authored or co-authored more than 60 refereed publications in the fields of software quality and metrics, computer dependability, maintenance resource allocation, air traffic control, and nuclear engineering.

References on page 5

Author Contact Information

Herbert Hecht
Chairman of the Board
SoHaR Incorporated
8421 Wilshire Blvd. #201
Beverly Hills CA 90211
Voice: (323) 653-4717 x110
Fax: (323) 653-3624

herb@sohar.com
www.sohar.com

Myron Hecht
President
SoHaR Incorporated
8421 Wilshire Blvd. #201
Beverly Hills CA 90211
Voice: (323) 653-4717
Fax: (323) 653-3624

myron@sohar.com
www.sohar.com

References

1. K. Kanoun and T. Sabourin, "Software Dependability of a Telephone Switching System", *Digest, Fault Tolerant Computing Symposium-17*, pp. 236-241, Pittsburgh, Pa., June 1987

2. General Accounting Office, High Risk Series: Information Management and Technology, GAO/HR97-9, Feb 97

3. D. E. Eckhardt, A. K. Caglayan, J. C. Knight, et al., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability", *IEEE Trans. Software Engineering*, vol 17 no 7, July 1991, pp. 692 - 702

4. B. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988, p.61

5. R. Balzer, N. Goldman and D. Wile, "Operational Specification as the Basis for Rapid Prototyping", *ACM Software Engineering Notes*, Dec 82, pp. 3 - 16

6. Mylopoulos, J (ed.), *Requirements Engineering*, IEEE Computer Society, 1997

7. Susan Gerhart, Dan Craigen and Ted Ralston, "Observations on Industrial Practice Using Formal Methods", *Proc. 15th International Conference on Software Engineering*, IEEE Computer Society Press, Baltimore, May 1993, pp. 24 - 33

8. D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of Safety-Critical Software", *Proc. Ninth Annual Software Reliability Symposium*, Colorado Springs CO, May 1991

9. Jarke, M., and R. Kurki-Suonio, eds. Special Issue on Scenario Management, *IEEE Transactions on Software Engineering*, vol 24 no 12, December 1998

10. Borgia, W. M., and N. J. Hrdlick,, "Thread-Based Integration Testing", *Software Tech News*, vol 3 no 3, (DACS), January 2000

11. Telford, D. G., "Task-Based Software Testing", *Software Tech News*, vol 3 no 3, (DACS), January 2000

12. P. G. Bishop, ed., *Dependability of Critical Computer Systems 3 - Techniques Directory*, Elsevier Applied Science, ISBN 1-85166-544-7, 1990

Using Failure History to Improve Reliability in Information Technology

Dolores R. Wallace, National Institute of Standards and Technology

Introduction

Achieving 100% software reliability may seem an unreasonable goal. Software developers and consumers of many software products are largely unsure about the reliability of their product or purchase. Today, many opportunities exist for some assurance of software products. Current practices and issues address process (e.g., CMM, ISO9000), people (e.g., software engineering degrees, certification exams, licensing) and product (e.g., measurement of the product); they encompass major areas of progress toward software reliability. One aspect of the product concerns the usage of history data of faults and failures of software systems, collected from either the development and assurance processes or operational use, to improve reliability of software products. Information contained in these histories characterizes the nature of faults, or defects, for a specific product line. The objectives are to use the history to determine how to prevent faults from entering into the product, to remove faults before the product is released, and to measure a product's frequency profile against others in the same domain. Finally, the histories may indicate problems indicating the need for better methods to prevent or detect faults, hence enabling justifiable research ideas.

Case Studies

Two case studies indicate how history data can be used. One is a study of failures of medical devices after release. The generic lessons learned here can be applied in other domains and the specific lessons may indicate how to study a domain and use failure history to ensure that reliable software is produced before the system is released.

Medical Device Failures

The medical device failures, involving no deaths or serious injury, occurred between 1983 and 1997. The 342 software-related failures were due principally to faults in logic, calculation, data, change impact, and others such as requirements, omission, quality assurance, timing, interface, initialization, configuration management and fault tolerance.

Many of these faults could have been prevented with requirements verification and with quality assurance practices aimed at the specific types of faults. For example, several of the calculation faults stemmed from using incorrect specifications for floating point calculations of the target computer or from not checking formula carefully against their source. Others, such as faults in logic with multiple conditions, indicate a need first for specifying the requirements correctly and second for finding methods for testing multiple conditions without exhaustive testing. In other instances, configuration management practices did not carefully retain version control for different international specifications. These are examples of specific lessons.

A different type of lesson is that knowledge of faults characteristic of a particular system aids decisions to direct verification and validation resources for optimum value. Specifically, the results indicate how failure data can be used to examine worst case scenarios in a product line and from there to identify how best to apply the quality practices to search for specific types of prevalent or characteristic faults for that product line. Results of this study provide an affirmation of generally accepted quality practices regardless of domain and may indicate the need for more sophisticated corrective techniques for solving requirements specification and logic problems.

Transportation Application

The second case study involves an application in transportation with data collected during the activities of requirements specification through acceptance testing of at least some parts of the system. In this instance, certain lessons had already been learned, that is, because most faults occur in the requirements specification, more effort should be expended to catch faults during that activity. The prevalent classes in this project are logic, specification, output, computation, performance, improvement, and initialization, with interface, omission, data handling, input, and documentation comprising the remaining groups. There was not enough information to classify approximately 10% of the faults.

Each fault is associated with its severity level and the development or test activity during which it was discovered. For example, 32% of the most severe problems were discovered during requirements specifications, but 31% were discovered during system test. Of all faults, 53% were discovered during requirements specification but 15% were caught during integration test, and 13% during system test. Why did those faults escape detection until so late, especially since 21 of those in system test had severity level 1? Could some of them have been detected with other activities during design and coding? Further examination of the fault classes detected in each group may lead to an understanding of issues and verification methods to be addressed better in the next similar project. More data are needed from additional projects to answer the types of questions posed by these two case studies.

NIST Repository and Support Tools for Collection of Data

One activity of the Error, Fault, and Failure Repository Project in the Information Technology Laboratory of the National Institute of Standards and Technology (NIST) is to collect additional data from projects. The objective of the project is to improve software quality by establishing fault models that reflect software failures in real-world systems. Greater understanding of the types of software errors that lead to faults and failures, along with the frequency and distribution of those faults, may result in the production of more reliable software. Project data from many domains will enable researchers and practitioners to understand weaknesses in current development and assurance methods and to affirm benefits of generally accepted quality practices. Some data may demonstrate the need for further research in selected topics, such as requirements specification and testing of complex systems. The NIST repository and support tools for collection of data are available to the public at

http://hissa.nist.gov/effProject/project.html

About the Author

Ms. Wallace leads the Reference Data: Software Fault & Failure Data Collection Project (http://hissa.ncsl.nist.gov/effProject) which provides metrology and reference data for software assurance. She is interested in methods of experimentation and measurement of software technology. Her publications on software verification and validation include NIST SP 500-234, Reference Information for the Software Verification and Validation Process, V&V articles in the Encyclopedia of Software Engineering (Wiley) and the IEEE Tutorials on Software Requirements Engineering and Software Engineering. She is co-author, Software Quality Control, Error Analysis, and Testing, Noyes Data Corporation, 1995 and co-chair of the IEEE STD 1012 -1998, Software Verification and Validation, and has published papers on software experimentation and other software engineering topics. She received the 1994 Department of Commerce Bronze Medal Award. Currently she serves on the editorial board of the American Society for Quality's Software Quality Professional and the Industrial Advisory Board for the IEEE Computer Society's Software Engineering Body of Knowledge Project. She has a master's degree in mathematics from Case Western University.

Author Contact Information

Dolores R. Wallace
National Institute of Standards and Technology
Information Technology Laboratory

dwallace@nist.gov
http://hissa.nist.gov/

A Software Engineering Course for Trustworthy Software

Continued from page 1

Often we do not know what load to expect. Dr. Vinton Cerf, inventor of the INTERNET, has remarked "applications have no idea of what they will need in network resources when they are installed." As a result, we try to avoid serious software problems by over engineering and over-testing. The Federal Food and Drug Administration of the USA notes:

> *"Software verification includes both static (paper review) and dynamic techniques. Dynamic analysis (i.e., testing) is concerned with demonstrating the software's run-time behavior in response to selected inputs and conditions. Due to the complexity of software, dynamic analysis alone may be insufficient to show that the software is correct, fully functional and free of avoidable defects. Therefore, static approaches and methods are used to offset this crucial limitation of dynamic analysis. Dynamic analysis is a necessary part of software verification, but static evaluation techniques such as inspections, analyses, walkthroughs, design reviews, etc., may be more effective in finding, correcting and preventing problems at an earlier stage of the development process."*

Software engineers cannot ensure that a small change in software will be limited to a small change in system performance. Industry practice is to test and retest every time any change is made in the hope of catching the unforeseen consequences of the tinkering. The April 25, 1994 issue of *Forbes Magazine* pointed out that a three-line change to a 2-million line program caused multiple failures due to a single fault. There is a lesson here. It is software failures, not faults that must be measured. Design constraints that can help software stability need to be codified before we can hope to deliver reliable performance. Instabilities arise in the following circumstances:

1. Computations cannot be completed before new data arrive,
2. Rounding-off errors build or buffer usage increases to eventually dominate system performance,
3. An algorithm embodied in the software is inherently flawed.

Trustworthy Software is now emerging as a technology area of interest. The IEEE is considering forming a special interest group, the Center for National Software Studies is sponsoring a web page devoted to exploring trustworthy software and the National Institute of Standards and Technology (NIST) gathers data on software defects and explores trustworthy issues.

A course dealing with these issues is being developed. It will feature design constraints that make software trustworthier. The topics planned include:

1. Case Histories of Failures.
2. Requirements Validation
3. Stability Analysis
4. Software Connectors
5. Ethics
6. Reliability Models
7. Failures, Faults and Defects
8. Testing
9. Software Visualization
10. Metrics

First Constraint: Software Rejuvenation

The first constraint limits the state space in the execution domain. Today's software runs non-periodically, which allows internal states to grow without bound. Software Rejuvenation is a new concept that seeks to contain the execution domain by making it periodic. An application is gracefully terminated and immediately restarted at a known, clean, internal state. Failure is anticipated and avoided. Non-stationary, random processes are transformed into stationary ones. One way to describe this is rather than running a system for one year with all the mysteries that untried time expanses can harbor, run it only one day, 364 times. The software states would be re-initialized each day, process by process, while the system continued to operate. Increasing the rejuvenation period reduces the cost of downtime but increases overhead. One system collecting on-line billing data operated for two years with no outages on a rejuvenation interval of one week.

A Bell Laboratories experiment showed the benefits of rejuvenation. A 16,000 line C program with notoriously leaky memory failed after 52 iterations. Seven lines of rejuvenation code with the period set at 15 iterations were added and the program ran flawlessly. Rejuvenation does not remove bugs; it merely avoids them with incredibly good effect.

Second Constraint: Software Fault Tolerance

If we cannot avoid a failure, then we must constrain the software design so that the system can recover in an orderly way. Each software process or object class should provide special code that recovers when triggered. A software fault tolerant library with a watchdog daemon can be built into the system. When the watchdog detects a problem, it launches the recovery code peculiar to the application software. In call processing systems this usually means dropping the call but not crashing the system. In administrative applications where keeping the database is key, the recovery system may recover a transaction from a backup data file or log the event and rebuild the database from the last checkpoint. Designers are constrained to explicitly define the recovery method for each process and object class using a standard library.

Third Constraint: Hire Good People and Keep Them

George Yamamura of Boeing's Space and Defense Systems reports that defects are highly correlated with personnel practices. Groups with 10 or more tasks and people with 3 or more independent activities tended to introduce more defects into the final product than those who are more focused. He points out that large changes were more error-prone than small ones, with changes of 100 words of memory or more being considered large. This may have some relationship to the average size of human working memory. The high .918 correlation between defects and personnel turnover rates is telling. When Boeing improved their work environment and development process, they saw 83 percent fewer defects, gained a factor of 2.4 in productivity, improved customer satisfaction and improved employee moral. Yamamura reported an unheard of 8 percent return rate when group members moved to other projects within Boeing.

Fourth Constraint: Limit the Language Features Used

Most communications software is developed in the C or C++ programming languages. Les Hatton's book, *Safer C: Developing Software for High-Integrity and Safety-critical Systems* (ISBN: 0-07-707640-0), describes the best way to use C and C++ in mission-critical applications. Hatton advocates constraining the use of the language features to achieve reliable software performance and then goes on to specify instruction by instruction how to do it. He says, "The use of C in safety-related or high integrity systems is not recommended without severe and automatically enforceable constraints. However, if these are present using the formidable tool support (including the extensive C library), the best available evidence suggests that it is then possible to write software of *at least* as high intrinsic quality and consistency as with other commonly used languages." For example, a detailed analysis of source code from 54 projects showed that once in every 29 lines of code, functions are not declared before they are used.

C is an intermediate language, between high level and machine level. There are dangers when the programmer can drop down to the machine architecture, but with reasonable constraints and limitations on the use of register instructions to those very few key cases dictated by the need to achieve performance goals, C can be used to good effect. The alternative of using a high level language that isolates the programmer from the machine often leads to a mix of assembly language and high level language code which brings with it all the headaches of managing configuration control and integrating modules from different code generators. The power of C can be harnessed to assure that source code is well structured. One important constraint is to use function prototypes or special object classes for interfaces.

Fifth Constraint: Limit Module Size and Initialize Memory

The optimum module size for the fewest defects is between 300 to 500 instructions. Smaller modules lead to too many interfaces and larger ones are too big for the designer to handle. Structural problems creep into large modules.

All memory should be explicitly initialized before it is used. Memory leak detection tools should be used to make sure that a software process does not grab all available memory for itself, leaving none for other processes. This creates gridlock as the system hangs in a wait state because it cannot process any new data.

Continued on page 10

A Software Engineering Course for Trustworthy Software

Continued from page 9

Sixth Constraint: Reuse Unchanged

A study of 3,000 reused modules showed that changes of as little as 10% led to substantial rework - as much as 60% - in the reused module. It is difficult for anyone unfamiliar with a module to alter it and this often leads to redoing the software rather than reusing it. For that reason, it is best to reuse tested, error-free modules as is.

Summary:

A course to codify and teach trustworthy software is proposed. The main idea is to design to avoid crashes and hangs so that software based systems become trusted systems. Trusted systems are those that repeatedly and reliably provide the same output for the same input when the environment is the same.

Suggestions and comments on course contents, approach and importance are sought.

About the Author

Mr. Bernstein is a recognized expert in Software Technology, project management, network management and technology conversion. He is president of the National Software Council with the goal of improving American software competitiveness, making software trustworthy and getting the software industry, the government and academia to work better together. He is now doing consulting through his firm Have Laptop - Will Travel and is the Executive Technologist with Network Programs, Inc. building software systems for managing telephone services.

Mr. Bernstein was an Executive Director of AT&T Bell Laboratories where he worked for 35 years.

As a software project manager he successfully built, sold and deployed a software system that automated the 100 million paper records telephone companies used to keep track of telephone lines to residential homes.

As a technologist he invented the concepts of 'dynamic provisioning' and 'routing to intelligence' which are included in the seven patents he holds. He saw the opportunities contained in research into software fault tolerance and championed its commercialization. It is now used in 24 products deployed in over 500 sites.

As a contributor to the profession he was recognized as a Fellow of the IEEE, the ACM and Ball State University. He is a member of the Russian based International Information Academy. He is a visiting associate of the Center for Software Engineering at the University of Southern California

Author Contact Information

Larry Bernstein
Have Laptop-Will Travel
4 Marion Ave.
Short Hills, NJ 07078-2120
(973) 258-9213

lberstein@worldnet.att.net

References:

1. Y. Huang, et. al. “Software Rejuvenation Analysis, Modules and Application,” *Proceedings of the 25th Symposium on Fault Tolerant Computing*, IEEE Computer Society June 27-30, 1995.

2. www.fda.gov - U.S. Food and Drug Administration Website

3. Jeffrey M. Voas, et.al. *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley and Sons, 1998, ISBN: 0-471-18381-4

4. John Musa, et al. *Software Reliability Measurement Prediction Application*, McGraw Hill 1987, ISBN 0-07-044093-X

5. Peter Neumann, *Computer Related Risks*, Addison-Wesley 1995, ISBN 0-201-55805-X

6. Robert L. Glass, *Software Reliability Guidebook*, Prentice Hall 1979, ISBN 0-13-821785-8

7. Tom Gilb, *Software Metrics* ISBN; 0-87626-855-6

8. John Stankovic, “Tutorial: Hard Real-Time Systems,” 1988 IEEE Computer Society, ISBN: 0-8186-0819-6

The SCR Requirements Method: A Practical Approach to Developing High Assurance Software Systems

by Constance Heitmeyer, Naval Research Laboratory

Background

In *high assurance systems*, compelling evidence is required that the system delivers its services in a manner that satisfies certain critical properties. Examples of high assurance systems include command and control systems, weapons systems, flight programs for commercial and military aircraft, control systems for nuclear plants, and most medical systems (e.g., patient monitoring systems). Critical properties high assurance systems must satisfy including *service* properties, i.e., properties of the services that the system delivers. For example, when a patient's heart stops, a patient monitoring system may be required to sound an alarm. Besides service properties, four other classes of critical system properties may be identified: *security* properties, *safety* properties, *real-time* properties, and *fault-tolerance* properties. In most cases, a high assurance system must satisfy properties in more than a single class. The patient monitoring system, for example, must satisfy not only service properties, but real-time constraints (the alarm must sound within some time interval), fault-tolerance properties (the alarm must sound even when a sensor malfunctions), and security properties (the system must protect the privacy of the patient's health records). Note that a given property may fall into more than a single class.

In recent years, researchers have proposed numerous approaches for specifying, constructing, and certifying high assurance systems. These include formal specification notations, formal models, and rigorous verification and validation techniques (such as model checking and mechanical theorem proving). But, two difficult problems remain. The first is the need for technology to support the application of these new techniques and methods to practical systems. Without such technology, opportunities to transfer many of the basic research results to industry are severely limited. Also needed is a unified framework for building systems that satisfy multiple critical properties. This need exists because not one but several different approaches for developing high assurance systems have evolved (at least one for each property class identified above). Each approach has a different overall philosophy of system development and different techniques for specification and assurance. No single one of the separate approaches is sufficient to handle systems now being built that must simultaneously satisfy properties in different classes. Thus a framework (i.e., formal specification techniques, formal models, assurance methods, etc.) is needed for constructing systems that must provide a set of critical services and do so in a secure, safe, timely, and fault-tolerant manner.

The SCR Requirements Method

One framework that has recently been developed for building *high assurance control systems* is based on the SCR (Software Cost Reduction) requirements method. Originally formulated in 1978 by Naval Research Laboratory (NRL) researchers to document the requirements of the flight program of the Navy's A-7E aircraft[5], SCR has been used in practice by numerous companies, including Grumann, AT&T, Ontario Hydro, and Lockheed, to specify software requirements. Designed for use by engineers, SCR has been applied to a wide range of systems, including telephone networks, control systems for nuclear plants, and both military and commercial flight control systems.

The SCR method offers a tabular notation for specifying requirements. Underlying this notation is a state machine model. Experience has shown that specifications in the SCR tabular notation are relatively easy for software developers to understand and to construct. Moreover, tables provide a precise, unambiguous basis for communication among developers and a natural organization for independent construction, inspection, modification, and mechanical analysis of parts of a large specification, and finally, the tabular notations scale. Evidence of the scalability of tabular specifications was demonstrated in 1993-94 by engineers at Lockheed, who, in the largest applications of the SCR method to date, used tables to specify the complete requirements of the C-130J flight program, a program containing over 250K lines of Ada code.

Continued on page 12

Continued from page 11

The SCR Toolset

Introduced in 1995, SCR[5,6] is an integrated suite of tools supporting the SCR requirements method. It includes a *specification editor* for creating a requirements specification, a *dependency graph browser* for displaying the variable dependencies in the specification, a *consistency checker* for detecting specification errors (e.g., type errors and missing cases), a *simulator* for validating the specification, and a set of *verification tools* for checking that the specification satisfies critical application properties, such as security and safety properties. All of the SCR tools are designed for ease of use. Moreover, to the extent feasible, the analyses performed by the tools are fully automatic. Except for the theorem prover (see below), use of the tools requires neither mathematical sophistication nor theorem proving skills. Currently, more than 100 academic, government, and industrial organizations in the US, the UK, Canada, Denmark, and Germany are experimenting with SCR. Further, many universities in both the U.S. and Canada include material on the SCR method and tools in their software engineering courses. The SCR tools are briefly described below.

Specification Editor. To create, modify, or display a requirements specification, the user invokes the specification editor. Each SCR specification is organized into dictionaries and tables. The dictionaries define the static information in the specification, such as the user-defined types and the names and values of variables and constants. Each table specifies how a given variable changes in response to an input event. One important class of tables specifies the values of the system outputs.

Dependency Graph Browser. Understanding the relationship between different parts of a large specification can be difficult. To address this problem, the Dependency Graph Browser (DGB) represents the dependencies among the variables in a given SCR specification as a directed graph. By examining this graph, a user can detect specification errors, e.g., undefined variables and circular definitions. The user can also use the DGB to display, extract, and analyze parts of the dependency graph, e.g., the subgraph containing all variables upon which a given system output depends.

Consistency Checker. The consistency checker[3] automatically detects syntax and type errors, variable name discrepancies, missing cases, nondeterminism, and circular definitions. When an error is detected, the consistency checker facilitates error correction by displaying the table (or dictionary) containing the error and highlighting the erroneous entries. It also provides an example that demonstrates the error. A form of static analysis, consistency checking is usually less expensive computationally than model checking. In developing an SCR specification, the user normally invokes the consistency checker first and postpones more expensive analysis, such as model checking, until later. Exploiting the special properties guaranteed by consistency checking (e.g., determinism) can make later analyses more efficient.

Simulator. To validate a specification, the user can run scenarios through the SCR simulator and inspect the results to ensure that the specification captures the intended behavior. Additionally, the user can define properties believed to be true of the required behavior and, using simulation, execute a series of scenarios to determine if any violate the properties. The SCR simulator also supports the construction of graphical front-ends, tailored to particular application domains. One example is a graphical front-end for pilots to use in evaluating an attack aircraft specification (see Figure 1). Rather than clicking on variable names, entering values for them, and seeing the results of simulation presented as variable values, a pilot clicks on visual representations of cockpit controls and views the results on a simulated cockpit display. This front-end allows the pilot to move out of the world of requirements specification and into the world of attack aircraft, where he is the expert. Clearly, a graphical interface facilitates validation of the specification.

Verification Tools. SCR provides three tools for checking that a requirements specification satisfies critical application properties. One tool is the model checker Spin. After a developer uses the SCR tools to generate a tabular

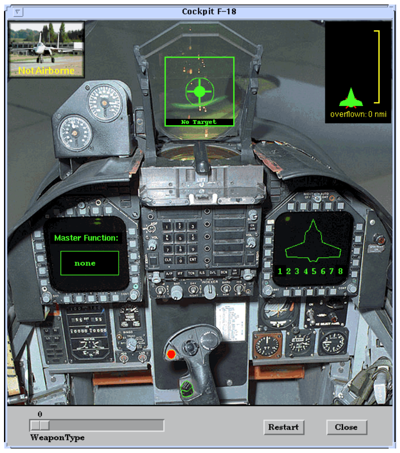


Figure 1. Customized Simulator Front-End for an Aircraft Specification

representation of the requirements, he can use SCR to automatically translate the tabular representation into the language of Spin. Then, the developer can invoke Spin to check that the specification satisfies properties of interest[5]. If Spin detects a property violation, the developer can then apply the SCR simulator to demonstrate and validate the violation. Because the state space of most specifications of practical systems is usually too large to analyze completely, model checking is typically used to *detect* violations of properties rather than to *verify* properties. To *verify* that an SCR specification satisfies selected properties, the developer may apply either of two additional tools. One tool called TAME[1] (Timed Automata Modeling Environment) is designed to facilitate mechanical theorem proving by reducing the human effort required to prove a property. TAME provides an interface to the mechanical proof system PVS. A second tool called Salsa[2] uses decision procedures and induction to verify selected properties, in many cases, automatically.

Practical Use of the SCR Tools

To date, the SCR tools have been applied in three pilot projects external to NRL. In the first, researchers at NASA's IV & V Facility used SCR to detect missing cases and instances of nondeterminism in the prose requirements specification of software for the International Space Station. In the second project, engineers at Rockwell Aviation used the tools to expose 24 errors, many of them serious, in the requirements specification of an example flight guidance system. Of the detected errors, a third were uncovered by entering the specification into the toolset, a third in running the consistency checker, and the remaining third in executing the specification with the simulator. In a third project, researchers at the JPL (Jet Propulsion Laboratory) used SCR to analyze specifications of two components of NASA's Deep Space-1 spacecraft for errors.

In addition, NRL has applied SCR to two military systems. In the first, NRL applied the SCR tools to a sizable contractor-produced requirements specification of the Weapons Control Panel (WCP) of a safety-critical weapons system[5]. The tools uncovered numerous errors in the contractor specification, including a safety violation that could result in the malfunction of a weapon. In a second project, NRL used the SCR tools to create and analyze an SCR specification of a moderately complex cryptographic device (CD) for a US Navy radio receiver[7]. Applying the SCR verification tools, demonstrated that the specification satisfies seven security properties but violates an eighth. Especially noteworthy is that, in each project, translating the original specification into the SCR notation, using SCR to analyze the specification for critical properties, and building a simulation (i.e., a working prototype of the system) required slightly more than one person-month. This small effort demonstrates the utility and cost-effectiveness of the SCR method.

Conclusions

The SCR tools can be distinguished in three major ways from commercial tools and other research tools. First, unlike most commercial tools, SCR has a formal foundation, thus allowing mathematically sound analyses (such as consistency checking and model checking), which are unsupported by most current CASE tools. Second, the SCR tools, unlike most research tools, have a well-designed user interface, are integrated to work together, and provide detailed feedback when errors are detected. Finally, users of SCR can perform significant analyses *without* detailed guidance from application experts or formal methods researchers, thereby providing formal methods usage at low cost.

Continued on page 14

The SCR Requirements Method

Continued from page 13

As demonstrated in the two NRL pilot studies, the SCR method has already been used to specify and to analyze both safety properties and security properties. The method also provides a means of specifying a system's real-time requirements; for example, in the case of the patient monitoring system, the requirement that "an alarm must be sounded within 0.1 seconds after a patient's heart has stopped". Analyzing such requirements specifications to ensure that they satisfy critical real-time properties is a topic of our current research. We are also exploring the use of the SCR method to specify and to analyze a system's fault-tolerance requirements, to construct test cases automatically, and to synthesize efficient source code from validated SCR requirements specifications.

About the Author:

Constance Heitmeyer heads the Software Engineering Section of the Naval Research Laboratory's Center for High Assurance Computer Systems. Since 1992, her section has been developing a collection of software tools to support the SCR requirements method. In 1996, she and D. Mandrioli published "Formal Methods for Real-Time Computing", a book whose purpose is to introduce practitioners to some of the most promising research results in real-time computing. Also, in 1996, Ms. Heitmeyer served as Program Chair for the COMPASS '96 Conference on Assured Computing. In 1997, she was General Chair of the 1997 International Symposium on Requirements Engineering. She has also served as an Associate Editor of the *Real-Time Systems Journal* and as a Guest Editor of a special issue of the *IEEE Transactions on Software Engineering*. Currently, she is a Guest Editor of a special issue of the Kluwer journal, *Formal Methods in System Design*, which is devoted to tabular expressions in software documentation and analysis.

Ms. Heitmeyer frequently teaches tutorials on the SCR approach to software development. Her interests are in requirements specification and analysis, formal methods, and real-time computing and in the transfer of research results into software practice.

For a copy of the SCR tools and the User Guide to the tools, send a request to C. Heitmeyer at her E-mail address.

Author Contact Information

Constance Heitmeyer
Naval Research Laboratory
(Code 5546)
Washington, DC 20375

Phone: (202) 767-3596
Fax: (202) 404-7942

heitmeyer@itd.nrl.navy.mil
www.chacs.itd.nrl.navy.mil/SCR/

References

1. M. Archer, C. Heitmeyer, and S. Sims, "TAME: A PVS Interface to Simplify Proofs for Automata Models," *Proceedings of User Interfaces for Theorem Provers (UITP '98)*, Eindhoven, Netherlands, July 1998.

2. R. Bharadwaj and S. Sims, "Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking," *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, March 2000.

3. C. Heitmeyer, R. Jeffords, and B. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Transactions on Software Engineering*, vol. 5, no. 3, July 1996, pp. 231-261

4. C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, "SCR*: A Toolset for Specifying and Analyzing Software Requirements," *Proceedings of Computer-Aided Verification (CAV'98)*, Vancouver, June-July, 1998.

5. C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, "Using Abstraction and Model Checking to Detect Safety Violations In Requirements Specifications," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, November 1998, pp. 927-948

6. K. Heninger, D. Parnas, J. Shore, and J. Kallander, "Software Requirements for the A-7E aircraft," Technical Naval Research Laboratory (NRL) report 3876, 1978.

7. J. Kirby, M. Archer, and C Heitmeyer, "SCR: A Practical Approach to Building a High Assurance COMSEC System," *Proceedings of 15th Annual Computer Security Applications Conference (ACSAC '99)*, December 1999.

Team Software Process Reliability Results

John B. Goodenough, Software Engineering Institute

Introduction

The Team Software Process (TSP)[1] defines how to create and manage software development teams. Software engineers who will become team members are first trained in how to do quality work, follow a defined process, and make process measurements that improve the quality of their work. This training is provided by an intensive 120-hour course that teaches the Personal Software Process^{SM} (PSP)[2]. The PSP course is very effective in teaching software engineers how to develop schedules they can meet, how to monitor their progress, and how to detect and remove defects in their code. The discipline followed by PSP-trained engineers enables high performance when working in teams, but individual process discipline is not enough. The Team Software Process provides the additional guidance individual engineers need to work effectively as part of teams.

With the TSP, each project starts with a *team launch* guided by TSP forms, scripts, and standards. The launch process is not a training exercise; team members work directly on their project, establishing team roles, team goals, detailed task plans (typically ten task-hours or less for the lowest level tasks), risks, assignments to tasks, and a presentation of the project's plan to management. The team meets weekly to measure progress and make any needed adjustments. Since the team has developed team goals and implementation plans, its members strive to meet the team's objectives.

The Team Software Process (TSP)

The Team Software Process has been developed by the Software Engineering Institute. The results presented here are drawn from 18 projects reported by four organizations: Boeing, Hill AFB, AIS (a small software company), and Teradyne. Data from the projects have been combined to illustrate team performance on several dimensions, before and after using TSP. Before/after ranges are shown for deviation from schedule, system test duration, acceptance test defects/KLOC, and post-release defects/KLOC. ("Before" data was not reported for every dimension by every organization.)

The effect of TSP on deviation from predicted schedule is striking. The organizations reporting schedule deviation data prior to their use of TSP said their average deviation from planned schedules ranged from 27% for the best reporting organization to 112% for the organization reporting the worst performance. For the 18 projects using TSP, the deviation from planned schedules ranged from –8% to +5%. The results are shown in Figure 1. This reflects a dramatic improvement in achieving planned schedules compared to the previous experience of the reporting organizations.

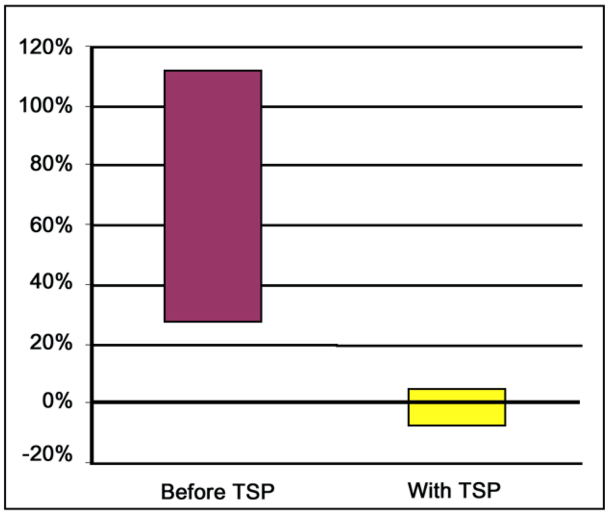


Figure 1. Schedule Deviation - Range

PSP training dramatically increases the ability of software engineers to detect defects injected in code prior to system test. Using the Team Software Process reinforces and encourages continued use of PSP techniques by all team members. Because defects are detected early and thoroughly, system test durations are drastically shortened, and the number of defects detected at acceptance test and after delivery are extremely low. As shown in Figure 2, system test duration decreased from an average of one to five days per KLOC to 0.1 to 1.0 days/KLOC. The number of acceptance test defects reported prior to use of TSP ranged from 0.1 to 0.7 defects per KLOC (see Figure 3). (The organization reporting 0.1 defects/KLOC was a CMM® Level 5 organization.) Acceptance test defects with the use of TSP ranged from 0.02 defects/KLOC (for the Level 5 organization) to 0.1 defects/KLOC (for all the other organizations, including the

Continued on page 16

Team Software Process Reliability Results

Continued from page 15

organizations operating at CMM Level 2). And finally, reported post-delivery defects (see Figure 4) ranged from zero to 0.1 defects/KLOC, a dramatic difference from the previous performance of these organizations.

In short, the investment in PSP and TSP training pays off in dramatically improved quality and dramatically improved ability to meet schedules, with improvements being exhibited for both low and high maturity organizations.

About the Author

John Goodenough is the Chief Technical Officer of the SEI and was named a fellow of the Association for Computing Machinery (ACM) in 1995. He is the former leader of the Rate Monotonic Analysis for Real-Time Systems Project. He was a Distinguished Reviewer for the Ada 95 language revision effort and has served as head of the U.S. delegation to the ISO Working Group on Ada. He was the principal author of the document specifying the revision requirements for Ada 95 and has served as chair of the group responsible for recommending interpretations of the Ada language.

Before joining the SEI, Goodenough was manager of the research and development department of SofTech, Inc. His work focused on the Ada programming language. He was the principal designer of one of the candidate languages leading to Ada. He later supported the Ada development effort as a distinguished reviewer for the Department of Defense, led the Ada Compiler Validation effort, and helped develop Ada training materials. Goodenough has worked at the Wang Institute of Graduate Studies as a visiting scholar, where he lectured on software reusability and testing and led seminars on object-oriented languages. He also has worked at the Air Force Electronic Systems Division in Bedford, Mass. There, he was responsible for formulating contract and in-house research and development, and he sponsored the first research work on software maintenance.

Author Contact Info

John B. Goodenough
Software Engineering Institute
Carnegie Mellon
Pittsburgh, PA 15213-3890

Phone: (412) 268-6391
Fax: (412) 268-5758

jbg@sei.cmu.edu
www.sei.cmu.edu/jbg/

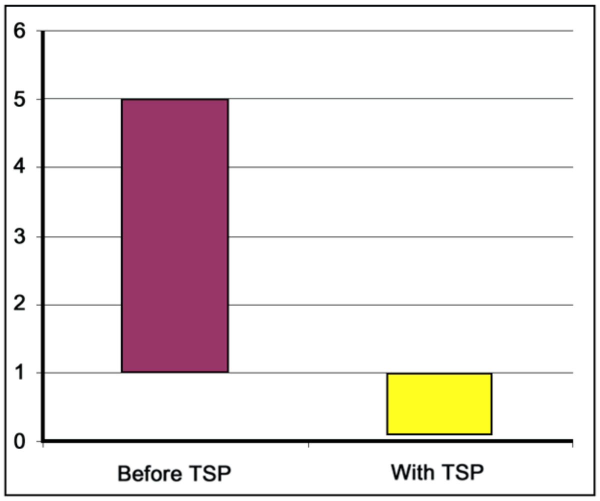


Figure 2. System Test Duration (Days/KLOC) - Range

Results from 18 projects executed by several organizations at different levels of process maturity show that use of the Team Software Process (TSP)SM yields dramatic improvement in the ability of software engineers to produce very high quality software, on time, at the predicted cost. A detailed presentation of these results is to be given at the Software Technology Conference in May 2000.

References

1. Humphrey, Watts S., *Introduction to the Team Software Process*, Addison-Wesley, Reading, MA, 2000.

2. Humphrey, Watts S., *A Discipline for Software Engineering*, Addison-Wesley, Reading, MA, 1995.

SM. Team Software Process, TSP, Personal Software Process, and PSP are service marks of Carnegie Mellon University. Capability Maturity Model and CMM are registered trademarks of Carnegie Mellon University.

Figures on page 17

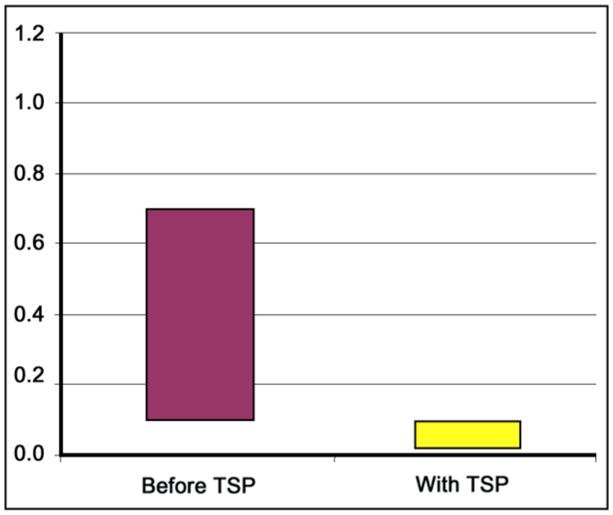


Figure 3. Acceptance Test Defects/KLOC

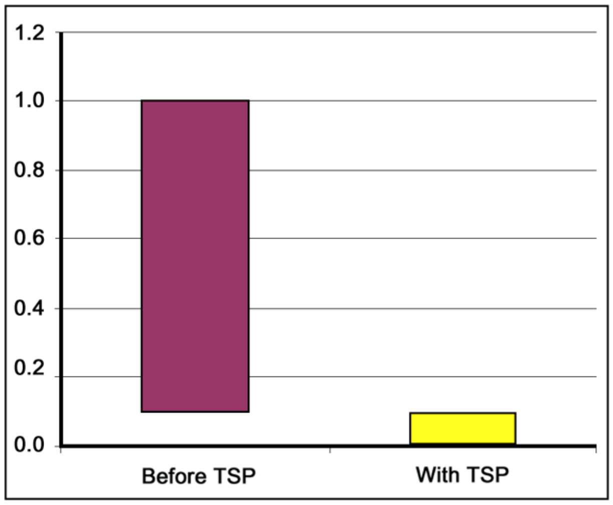


Figure 4. Post-Release Defects/KLOC - Range

Announcement: IAC Awareness Conference

"Key Challenges"

The Defense Technical Information Center Information Analysis Center Program Management Office will sponsor an IAC Awareness Conference on May 16, 2000 at the Hope Hotel, Wright Patterson Air Force Base, Dayton, Ohio.

Conference Theme and Objective

The *theme* of this conference is "Key Challenges" that need to be conquered to enable us to meet Vision 2010.

The *objective* of this conference is to explore the strategic direction and the resulting requirements of information technology and services necessary to support the DoD. To that end, an aggressive agenda with senior-level participants will provide an opportunity to discuss and share valuable insights between Research and Development and the warfighter community.

Who Should Attend?

The meeting is open to all Department of Defense (DoD) and associated industry personnel. This meeting will promote IAC Awareness with an emphasis on the needs of the warfighter. Those in attendance will include policy makers, DoD program managers, researchers, analysts, information providers, and information users. This conference will address the information needs of the warfighter, along with the current and future information technology initiatives to support those needs in the new millennium. The impact of changes in the policies, procedures, and technologies of information now and in the future and the subsequent impact on DoD will also be addressed.

DoD IACs will have exhibits in the display area highlighting their capabilities, products, and services.

May 16, 2000

Hope Hotel
Wright Patterson AFB
Dayton, Ohio

Registration

Electronic Registration is encouraged. Register on-line at: http://iac.dtic.mil/surviac

Or Contact:

Donna Egner at SURVIAC

Phone: (937) 255-4840
Fax: (937) 255-9673

degner@bah.com.

Practical Software and System Availability and Reliability Estimation

by John Gaffney, Lockheed Martin

Introduction

In an increasing number of cases, the acquirers of software intensive systems for both Government and civilian applications are requiring that these systems meet certain availability and/or reliability objectives. This article presents aspects of a method and supportive tools that have been developed and applied at Lockheed Martin in the estimation of availability and reliability of various large software intensive systems. The methodology and supportive tools have evolved from work started at the former IBM Federal Systems, now various Lockheed Martin companies.

Availability and Reliability

Availability and reliability are closely related measures. A definition for availability is the proportion of some period of time that the system is operating satisfactorily. What satisfactorily means must be defined, of course. A definition for reliability is the probability that the system does not fail for a stated length of time, starting from some specific time. For calculations/estimates of availability, we need information about both failure times/rates and time/rates for service restoration (not necessarily repair). We have used the term system in the definitions just presented. We could also apply them to the hardware or software components of a system, such as a software intensive system. We can combine the unavailabilities for a system due to software or due to hardware (and to procedures on some occasions) into an overall figure for system unavailability; availability is equal to one minus unavailability.

There are two principal types of availability and reliability models, the *black box* and the *white box*. The former requires detailed knowledge of the (hardware and software) elements of the system of concern. The latter looks at the externally visible behavior of the system, without requiring knowledge of the detailed nature of such elements or their interactions (such as those among failure tolerant software and/ or hardware units which might mask the effects of certain types of hardware or software failures from the external visible behavior of the system)

The principal parameter to be estimated for hardware or software failure is λ, the failure rate (such as failures per hour, month, etc.). In the case of software, this is actually a function of time, as the failure rate eventually decreases as defects are discovered and few if any new defects (not present before testing began) are added. In the case of hardware, we often consider λ to be fixed; that is, the failure rate for the hardware or for hardware-caused system outages (under the white box model as defined above) is constant because the hardware is assumed to have "burned in" or is mature or has a mature design. We now focus on software as it has a non-constant λ. After the software has been delivered (i.e., placed in operation), the function $\lambda(t)$ is often taken to be a monotonically decreasing function of time, say of the form:

$$\lambda(t)= \lambda_0*(\exp(-bt)),$$

where λ_0=E*b, E=the number of defects in the software at t=0 (i.e., at delivery/start of operation) and $b=1/t_p$, where t_p, the time constant, is the time at which 0.63*E defects will have been discovered. In actuality, this is often not the case for a period of time after delivery of the software/when it is placed in operation. During that period, there is often a "surge" of defect discovery. This is due to one or both of two principal causes, the test environment did not completely accurately represent the operational environment (and there were different stimuli of types not applied during testing), and/or there was additional software, not present in the test suite, that opened other error paths.

We can obtain an estimate for E in various ways. The most accurate is to fit actual defect discovery data, obtained during the development and testing process and fit it to an assumed equation, say using the STEER model tool, the original version of which was developed at IBM, Federal Systems. If both integration and testing are considered, the form of the time-based model of defect discovery is a Rayleigh curve (or another, similar monomodal-single-peak-curve of the Weibull family). Then, an estimate for E would be the area under the right hand side of the monomodal curve from the time of the delivery of the software (or when it is placed in operation) to infinity. Or, alternatively, an exponential fit can be made to the defect discovery data obtained from the time of the peak forward. That is, the portion of the curve to the right of the peak can be approximated as an ("decaying") exponential, which is the form given above for λ(t). Then, the values for the parameters E and b (=1/t_p) for this curve can be used to estimate λ(t) in the vicinity of some value for t, t=t_n, where it is desired to compute the software reliability or availability or the unavailability of the system due to software-caused failures.

About the Author

John Gaffney is a Software Engineering Consultant, at Lockheed Martin, Mission Systems in Rockville, MD. He provides support to Lockheed Martin organizations in software and systems measurement. Earlier, he worked at the Software Productivity Consortium where he started the measurement program. Previously, he worked at IBM. He has taught at Polytechnic University, Brooklyn, NY and at Johns Hopkins. He holds a BA from Harvard, and MS from Stevens Institute of Technology, and is a Registered Professional Engineer (Electrical) in the District of Columbia.

Author Contact Info

John Gaffney Jr.
Lockheed Martin
Mission Systems and Software Resource Center
9211 Corporate Boulevard
Rockville, MD 20850

Phone.: (301) 640-2359
Fax: (301) 640-2429

j.gaffney@lmco.com

DACS Product Announcement:
The DACS Software Reliability Sourcebook

Introduction

Over the past several years, the phenomenal growth of software reliability engineering has resulted in hundreds of hardcopy and on-line resources for both researchers and practitioners.

There appears to be a need, therefore, to bring together basic software reliability techniques and available resources for additional information into one concise handbook that can be used to meet the day-to-day requirements of the practitioner while still satisfying those wishing to expand their understanding of the subject matter. Development and publication of The *DACS Software Reliability Sourcebook* is intended to meet that need.

Description

The *DACS Software Reliability Sourcebook* is intended to be a resource for software developers, testers and managers that will include, as a minimum:

- ❑ Basic mathematical concepts associated with software reliability engineering
- ❑ Basic software reliability definitions and metrics

In addition to these basic concepts and definitions the following areas of the DACS Software Reliability Sourcebook will have an additional DACS URL location that will cross-reference to applicable Websites and keep these references up-to-date in "real-time".

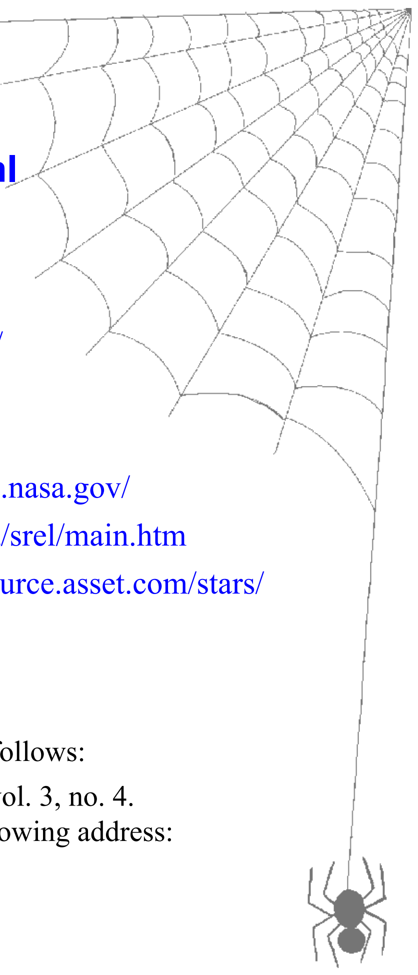
- Concise descriptions and examples of the most popular or promising software reliability engineering models, analytical techniques and test practices
- Descriptions of appropriate automated tools
- Bibliographic references, where readers can go for additional information
- On-line references, where readers can go for additional information
- Identification of current research initiatives and areas of software reliability that require further study
- Identification of COTS software reliability issues and resources
- Concise descriptions of current standards and specifications that deal with software reliability, including ordering information from the appropriate standards organizations

Material for this publication will be developed from the available published literature, Internet on-line searches, internal DACS software reliability engineering expertise, and the DACS Expert Network.

This Product Will Be Available, June 2000.

Software Tech News on the World Wide Web

This newsletter, including referenced full-length articles, is available on the web at:
www.dacs.dtic.mil/awareness/newsletters/listing.shtml



Other Software Reliability On-line Resources

DACS Software Reliability Topic Area - www.dacs.dtic.mil

High Integrity Software System Assurance (HISSA) - http://hissa.nist.gov/

Quanterion Solutions - www.acq.osd.mil/te/

Reliability Analysis Center (RAC) - http://rac.iitri.org

Software Assurance Technology Center (SATC) at NASA - http://satc.gsfc.nasa.gov/

Software Reliability Engineering Information Center - www.enre.umd.edu/srel/main.htm

Software Technology for Adaptable, Reliable Systems (STARS) - http://source.asset.com/stars/

Software Testing Institute - www.ondaweb.com/sti/

Article Reproduction

DoD Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

First-Class Mail
U.S. Postage
P A I D
Colo. Spgs., CO
Permit No. 745

Return Service Requested

DoD DACS Products & Services Order Form

Name: | Position/Title:

Organization: | Acronym:

Address:

City: | State: | Zip Code:

Country: | E-mail:

Telephone: | Fax:

Product Description	*Format*	*Quantity*	*Price*	*Total*
The DACS Information Package	*Note: All Disks are available in PC or Mac			
❑ Including: 2 recent Software Tech News newsletters, and several DACS Products & Services Brochures	Documents		FREE	FREE
Empirical Data				
❑ Architecture Research Facility (ARF) Error Dataset	Disk		$ 50	
❑ Goel-Okumoto Software Reliability Model 1.0				
❑ NASA / Software Engineering Laboratory (SEL) Dataset	CD-ROM		$ 50	
❑ NASA / AMES Error/Fault Dataset	Disk		$ 50	
❑ Software Reliability Dataset	Disk		$ 50	
❑ DACS Productivity Dataset	Disk		$ 50	
Technical Reports				
❑ A Business Case for Software Process Improvement (Revised)	Document	FREE with Spreadsheet →	$ 25	
❑ Measuring ROI from Software Eng Management Spreadsheet	CD-ROM		$ 50	
❑ A History of Software Measurement at Rome Laboratory	Document		$ 25	
❑ An Analysis of Two Formal Methods: VDM and Z	Document		$ 25	
❑ An Overview of Object-Oriented Design	Document		$ 25	
❑ Artificial Neural Networks Technology	Document		$ 25	
❑ A Review of Formal Methods	Document		$ 25	
❑ A Review of Non-Ada to Ada Conversion	Document		$ 25	
❑ Software Design Methods	Document		$ 25	
❑ Distributable Database Technology	Document		$ 25	
❑ Mining Software Engineering Data: A Survey	Document		$ 50	
❑ Object Oriented Database Management Systems (Revisited)	Document		$ 50	
❑ Software Analysis and Testing Technologies	Document		$ 25	
❑ Software Design Methods	Document		$ 25	
❑ Software Prototyping and Requirements Engineering	Document		$ 25	
❑ Software Interoperability	Document		$ 25	
❑ Software Reusability	Document		$ 25	
❑ Understanding & Improving Technology Transfer in Soft Eng	Document		$ 50	
❑ Using Defect Tracking & Analysis to Improve Software Qual	Document		$ 50	
❑ DACS Technical Reports CD (Includes reports listed above)	CD-ROM		$ 200	
Bibliographic Products				
❑ Rome Laboratory Research in Software Measurement	Document		$ 25	
❑ DACS Custom Bibliographic Search	Disk		$ 40	
❑ DACS Software Engineering Bibliographic Database (SEBD)	CD-ROM		$ 50	

Method of Payment:

❑ Check ❑ Mastercard ❑ Visa

Number of Items Ordered ☐ *Total Cost* ☐

Credit Card # ____________________ Expiration Date ____________________

Name on Credit Card ____________________ Signature ____________________

Mail this form or: Phone: (800) 214-7921, Fax: (315) 334-4964
E-mail: **cust-liasn@dacs.dtic.mil**

This form is also on-line at: www.dacs.dtic.mil/forms/orderform.shtml

---fold here---

---fold here---

Fix
postage
here

DoD Data & Analysis Center for Software
Attn: DACS Customer Liaison
PO. Box 1400
Rome, NY 13442-1400